Les microcontrôleurs PIC programmés en Objective Caml

Benoît Vaugon[1], **Philippe Wang** [2], **Emmanuel Chailloux** [2]

1: Université Pierre et Marie Curie (Paris 6)
Benoit.Vaugon@etu.upmc.fr

2: Laboratoire d'informatique de Paris 6 (LIP6), CNRS UMR 7606, Université Pierre et Marie Curie (Paris 6)
Philippe.Wang@lip6.fr
Emmanuel.Chailloux@lip6.fr

Résumé. *Les microcontrôleurs PIC sont des circuits intégrés programmables. Ils se distinguent par leurs bas prix et leur faible consommation énergétique. Les principaux langages de programmation disponibles sont l'assembleur PIC et des sous-ensembles des langages C et Basic. Ils demandent souvent au programmeur de se préoccuper méticuleusement des limitations du matériel. Dans ces conditions, il est souvent difficile d'exprimer des algorithmes complexes utilisant des allocations dynamiques.*

Nous introduisons Objective Caml, un langage applicatif multiparadigme, à allocation dynamique et récupération automatique de mémoire. L'utilisation d'un langage de haut niveau sur ces architectures apporte la possibilité d'y exécuter des algorithmes bien plus complexes tout en assouplissant fortement les contraintes de programmation, notamment parce que la gestion de la très petite quantité de mémoire est alors automatisée. Cela facilite le processus de mise au point tout en garantissant un haut niveau de sûreté d'exécution.

Pour ce faire, nous introduisons une machine virtuelle Objective Caml développée en assembleur PIC18, selon les spécifications de la machine de référence distribuée par l'INRIA. Celle-ci permet d'utiliser la totalité du langage Objective Caml sans aucune restriction. Les performances obtenues sont par ailleurs très satisfaisantes et soutiennent la pertinence de l'approche adoptée.

Mots-Clés : *Microcontrôleur PIC18, Objective Caml, Machine Virtuelle*

1. Introduction

Lors de la conception d'un circuit électronique, si celui-ci nécessite une unité de calcul, l'implantation de celle-ci est soit un assemblage de portes logiques (programmation matérielle), soit un microcontrôleur (programmation logicielle). Les premiers ont un très faible coût de fabrication s'ils sont produits en très grande quantité. L'avantage des seconds est qu'ils sont de toute manière fabriqués en masse afin de les rendre le plus accessible possible en réduisant au maximum leur prix, et il suffit d'y embarquer un logiciel pour qu'ils puissent accomplir une tâche spécifique.

Les PIC [1, 8] sont des microcontrôleurs programmables disponibles à bas prix, commercialisés par la société Microchip[1]. Leur facilité d'intégration dans un petit circuit électronique fait qu'ils sont notamment très utilisés par les « bricoleurs en électronique », en plus des utilisateurs industriels (*e.g.*, robots électroménagers). Ils intègrent une unité de calcul, une mémoire de calcul volatile et une mémoire de masse non volatile. Toutes trois ont une capacité très limitée, en comparaison avec ce qui se fait par ailleurs en « matériel informatique grand public ». En effet, avec l'unité de calcul cadencée à une fréquence de quelques MHz, selon les séries et modèles adoptés, le nombre d'instructions exécutées par seconde varie entre 1 et 40 millions. La mémoire de calcul a une capacité incluse entre 16o et 128ko. La mémoire de masse a une capacité comprise entre moins d'un ko et 512 ko. La taille des mots machine dépend des séries de PIC ; celles disponibles sont le plus souvent 8 bits (séries PIC10, PIC12, PIC16 et PIC18), 16 bits (séries PIC24, PIC30, PIC33) ou 32 bits (série PIC32) pour les plus récents et puissants. On peut remarquer que cette capacité de calcul est comparable à celles de microprocesseurs des années 1980 (*e.g.*, Intel 8080, Zilog Z80).

Ces puces sont habituellement programmées en assembleur directement ou bien à l'aide d'un sous-ensemble de Basic ou C. Cependant, le jeu d'instructions assez particulier des PIC rend la génération de code PIC assez difficile. Par exemple, il n'y a pas d'ins-

1. `http://www.microchip.com`

truction matérielle pour la division de nombres entiers sur la série PIC18, et les instructions pour la multiplication sont absentes sur les séries antérieurs. De nombreux compilateurs n'apportent que des commodités minimales tant au niveau des constructions de langage prises en charge qu'au niveau de la bibliothèque standard. La faible quantité de ressources rend leur programmation assez délicate et parfois très ardue, notamment lorsqu'il s'agit d'utiliser des algorithmes complexes sur des structures de données dynamiques.

Nous proposons d'utiliser un langage applicatif riche, non seulement fonctionnel mais aussi modulaire, impératif et à objets, statiquement typé, à gestion de mémoire automatique, afin de faciliter la programmation de telles puces et de rendre les logiciels plus sûrs. Nous introduisons ainsi Objective Caml [7] dans le monde des outils dédiés à la programmation des microcontrôleurs PIC.

Objective Caml est développé et distribué par l'INRIA avec deux compilateurs. Le premier émet du code-octet pour une machine virtuelle, appelée ZAM [6] par la suite, incluse dans la distribution. Le second émet du code pour le processeur matériel de la machine réelle qui hébergera les programmes ainsi compilés. Les deux compilateurs partagent une bibliothèque d'exécution implantée en C.

En se basant sur l'existant, pour pouvoir exécuter sur un PIC un programme écrit en Objective Caml, nous avons eu à choisir entre compléter le générateur de code du compilateur natif pour cibler les machines PIC ou bien fournir une machine virtuelle fonctionnant sur PIC pour interpréter les programmes compilés en code-octet.

Modifier l'un des deux compilateurs implique un travail de maintenance conséquent *a posteriori* du fait de l'évolution de la distribution d'Objective Caml. Et l'émission de code PIC étant assez difficile, l'idée d'étendre le compilateur dans cette optique a été assez vite abandonnée, au profit de l'implantation d'une machine virtuelle sur PIC pour Objective Caml.

Cet article présente notre réalisation, appelée OCAPIC[2], qui répond à la question : comment faire tenir une machine virtuelle pour Objective Caml sur un PIC ?

Dans la suite de cet article, nous commençons par décrire les caractéristiques des microcontrôleurs PIC plus en détail afin de mieux comprendre la problématique des contraintes du matériel. Puis, les sections 3 et 4 présentent la machine virtuelle Objective Caml et son implantation sur un PIC. La section 5 propose plusieurs moyens de réduction de la taille des programmes, afin de pouvoir en embarquer davantage sur les PIC. L'environnement de développement fourni avec la distribution est ensuite détaillé en section 6. Puis, une étude de performances est décrite en section 7. Enfin, nous discutons des différentes expérimentations de programmation de PIC dans des langages de haut niveau pour les comparer avec la nôtre en section 8 et concluons sur notre expérience en section 9.

2. Les microcontrôleurs PIC

2.1. *Les microcontrôleurs*

Les microcontrôleurs sont des circuits intégrés programmables. Ils regroupent, au sein d'une même puce, les différents éléments que l'on trouve habituellement dans l'unité centrale d'un ordinateur personnel. En particulier, ils contiennent une unité de calcul, différentes mémoires volatiles et non volatiles, ainsi qu'un ensemble « d'interfaces internes » facilitant la communication avec le monde extérieur. Ils sont conçus pour être programmés par un ordinateur, puis placés dans un circuit électronique dans lequel ils effectuent un travail plus ou moins complexe. Après programmation, ils peuvent fonctionner indépendamment de tout ordinateur.

Leurs principaux domaines d'application sont l'industrie pour la fabrication de machines outils, la domotique (machines à café, appareils électroménagers, etc.), et l'électronique grand public.

2. http://www.algo-prog.info/ocaml_for_pic/

Différents exemples d'applications sont détaillés dans des revues d'électronique, comme par exemple la revue Elektor[3]. Leur production en grande série fait baisser leur prix de revient (en général de l'ordre de quelques dollars américains). Ceci les rend intéressants pour les amateurs comme pour les industriels.

Ils peuvent dans beaucoup de cas remplacer les architectures dédiées, c'est-à-dire les circuits intégrés conçus matériellement pour une tâche précise. Il est en effet généralement plus facile de concevoir et de maintenir un logiciel qu'un circuit électronique. Les architectures dédiées sont cependant parfois nécessaires pour des raisons de performances.

Différentes familles de microcontrôleurs sont disponibles sur le marché. Parmi les plus connus, on trouve le 6800 et ses dérivés (comme le 68HC11) produits à l'origine par la société Motorola puis par Freescale Semiconductor ; les microcontrôleurs Philips (en particulier les séries de P87 et P89) ; ainsi que les PIC de la société Microchip.

2.2. *Les PIC*

Les PIC forment une famille de microcontrôleurs développés par la société Microchip. Il existe plusieurs centaines de modèles regroupés dans différentes séries (PIC16, PIC18, etc.) en fonction de leurs caractéristiques. Leurs prix varient entre approximativement 0,30 et 9 dollars américains d'après le site de Microchip.

Chaque série de PIC possède son propre langage assembleur. Les programmes écrits pour une série ne sont en général pas compatibles pour les autres. Cependant, il est souvent facile de porter de manière quasi systématique les programmes d'une série de numéro faible vers une série de numéro plus grand.

Les PIC possèdent une architecture RISC. Leurs jeux d'instructions se composent d'instructions arithmétiques et logiques, d'instructions de branchement, et de quelques instructions spéciales

3. `http://www.elektor.fr/`

permettant par exemple d'accéder à la mémoire contenant le programme. Voici un petit extrait de code assembleur PIC :

```
exemple_de_code:          ; label
     tblrd*+              ; lecture dans la mémoire flash
     movf    TABLAT, W    ; transfert d'un registre vers W
     addwf   FSR0L, W     ; addition du registre FSR0L et W
     rcall   SOUS_PRGM    ; appel au sous programme SOUS_PRGM
     btfsc   STATUS, C    ; saut de l'instr suivante si CARRY=0
     incf    FSR0H, F     ; incrémentation de FSR0H
     return               ; retour à l'appelant
```

Les PIC possèdent en général quatre mémoires distinctes. L'une est une mémoire non-volatile utilisant la technologie flash, appelée « mémoire programme ». Comme son nom l'indique, elle contient le programme à éxécuter. La programmation d'un PIC consiste donc en l'écriture du programme dans cette mémoire. Elle est en général réinscriptible de 1 000 à 100 000 fois selon les PIC.

Pour stocker dynamiquement des informations, les PIC utilisent une mémoire constituée uniquement de registres. Le nombre de registres varie selon les PIC entre quelques dizaines, jusqu'à plus d'une centaine de milliers. Ils sont accessibles directement par l'unité de calcul, en lecture et en écriture.

Dans les anciennes séries de PIC, la mémoire programme n'est pas accessible en écriture depuis le PIC, mais uniquement à la programmation. Il est cependant parfois nécessaire de stocker des informations devant être conservées lorsque le PIC n'est plus alimenté électriquement. Pour ce faire, certains PIC possèdent une EEPROM[4] de faible capacité, accessible en lecture et en écriture depuis le PIC. Ces mémoires sont en général réinscriptibles entre 1 000 000 et 10 000 000 de fois.

Les PIC possèdent un mécanisme d'appel de procédure. Il est basé sur les instructions assembleur `call` et `return` permettant respectivement d'appeler une procédure et de retourner à l'appelant. Cependant, pour la plupart des séries de PIC, la taille des

4. Electrically-Erasable Programmable Read-Only Memory ou mémoire morte programmable effaçable électriquement.

registres ne suffit pas pour y stocker des adresses de la mémoire programme. Les adresses de retour sont alors stockées dans une pile particulière. La mémoire constituant cette pile est organisée en mots dont la taille ne dépend pas de l'architecture de l'unité de calcul, mais de la taille de la mémoire programme. La taille de ces mots est par exemple de 14 bits pour les PIC de la série PIC16.

Le tableau 1 donne, pour chaque série de PIC, les valeurs extrêmes de leurs principales caractéristiques techniques.

Famille	Archi[bit]	Flash[ko]	Registres[o]	EEPROM[o]	MIPS
PIC10	8	.375 → .75	16 → 24	0	1 → 2
PIC12	8	.75 → 7	25 → 256	0 → 256	1 → 8
PIC16	8	.75 → 28	25 → 1536	0 → 256	4 → 8
PIC18	8	4 → 128	256 → 4096	0 → 1024	8 → 16
PIC24F	16	4 → 256	512 → 98304	0 → 512	16
PIC24H	16	12 → 256	1024 → 16384	0	40
dsPIC30	16	6 → 144	256 → 8192	0 → 4096	30
dsPIC33	16	6 → 256	256 → 30720	0	30
PIC32	32	32 → 512	8192 → 131072	0	10 → 20

Tableau 1 – Tableau comparatif des différentes séries de PIC

Les PIC ne possèdent pas d'instructions spécifiques pour communiquer avec l'extérieur, ni pour configurer les interfaces internes. Ceci se fait en lisant et en écrivant dans des registres particuliers du PIC. Les pattes du PIC sont regroupées en « ports », et chaque port correspond à un registre du PIC. Par exemple, pour appliquer une tension de 5V sur la patte numéro 3 du port B, il suffit de mettre à 1 le bit numéro 3 du registre PORTB. Ceci peut se faire par l'instruction assembleur :

```
1         bsf     PORTB, 3
```

De même, pour configurer l'interface EUSART[5] permettant de communiquer avec un port série d'un ordinateur, il suffit d'écrire dans le registre `TXSTA`.

L'une des difficultés du portage d'un langage de haut niveau comme Objective Caml sur PIC est de permettre au programmeur

5. Enhanced Universal Synchronous/Asynchronous Receiver Transmitter.

d'effectuer ce genre d'opérations aussi facilement qu'en assembleur, sans pour autant casser les principes du langage. Il faut alors donner au programmeur des moyens élégants et performants d'effectuer des opérations dont la sémantique est une écriture dans la mémoire à une adresse fixée.

2.3. *Le PIC18F4620*

Dans notre réalisation, le code assembleur a été écrit pour la série PIC18. Cette série a la particularité d'avoir un « jeu d'instructions étendu » créé à l'origine pour faciliter la compilation depuis le langage C. Ces instructions nous ont en particulier permis de faciliter la gestion de la pile Objective Caml et d'améliorer significativement les performances.

Plus précisément, le PIC sur lequel les tests ont été effectués est le PIC18F4620. Il a été choisi principalement pour son grand nombre de registres. En effet, la quantité de mémoire dynamique est le facteur qui nous a posé le plus de difficultés pour porter des programmes Objective Caml sur PIC. Ses principales caractéristiques sont présentées dans le tableau 2.

Architecture	8 bits
Mémoire flash	64 ko
Registres	3968 o
EEPROM	1024 o
MIPS	10
Entrées/sorties	36 broches

Tableau 2 – Caractéristiques du PIC18F4620

3. La machine virtuelle Objective Caml

La machine virtuelle Objective Caml est une machine à pile. Elle permet d'exécuter le code-octet généré par le compilateur `ocamlc`. Un tel exécutable contient principalement trois sections :

– CODE : contient le code-octet.

– DATA : contient la sérialisation d'un tableau de variables globales. Ce tableau contient des valeurs précalculées par le compilateur (par exemple des flottants, des chaînes de caractères, des exceptions), ainsi que des cases non-initialisées qui seront affectées à l'exécution.

– PRIM : contient les noms de fonctions externes. Il associe un numéro à chaque fonction externe pour y accéder rapidement.

D'autres sections sont parfois présentes. Elles permettent de repérer des bibliothèques dynamiques (DLLS et DLPT), ajouter des informations de débogage (DBUG), ou autre. Il est difficile de leur donner un sens sur PIC, nous avons donc décidé de les ignorer lors du portage.

À l'initialisation, la machine virtuelle désérialise le tableau des variables globales, associe chaque fonction externe avec son numéro, et démarre l'exécution du code-octet. Lors de l'exécution, elle gère différents éléments :

– *stack* : une pile d'évaluation.

– *accu* : un accumulateur pouvant être vu comme le sommet de la pile.

– *heap* : un tas contenant les valeurs allouées.

– *code* : un segment contenant le code-octet à exécuter.

– *pc* : un compteur ordinal pointant sur le segment contenant le code-octet.

– *data* : un tableau contenant les valeurs des variables globales.

– *env* : une variable pointant, lors de l'évaluation du corps d'une fonction, vers la fermeture associée. Celle-ci contient entre autres les valeurs des variables libres du corps de la fonction, ce qui permet d'y accéder rapidement.

– *trapSp* : un registre stockant la position dans la pile du dernier rattrapeur d'exception posé.

– *extraArgs* : un compteur stockant, lors d'un appel de fonction, le nombre d'arguments passés. Il est utilisé pour repérer les appels partiels.

Il existe 146 instructions de code-octet différentes. Cependant, 60% sont des alias permettant de factoriser le code et d'améliorer la vitesse d'exécution. Une instruction de code-octet est composée d'un identifiant d'instruction (opcode) et éventuellement d'arguments. Dans l'exécutable généré par `ocamlc`, chaque opcode et chaque argument sont codés sur quatre octets. Ces instructions peuvent être groupées en sept catégories :

– calculs arithmétiques et logiques.

– contrôle : branchements conditionnels et inconditionnels.

– gestion de la mémoire : accès à la pile et au tas, allocations dynamiques.

– gestion des fermetures : créations de fermetures, appels et retours de fonctions.

– gestion des exceptions : poses de rattrapeurs, lancements d'exceptions.

– gestion des objets : appels de méthodes.

– gestion des appels aux fonctions externes.

On peut se référer à la documentation de X. Clerc concernant le projet Cadmium [3] pour le détail de chaque instruction.

4. L'implantation de la machine virtuelle Objective Caml sur PIC

Cette section décrit l'implantation que nous avons réalisée pour permettre d'exécuter des programmes Objective Caml sur les PIC. Elle détaille les techniques que nous avons utilisées pour implanter l'interprète de code-octet et la bibliothèque d'exécution, ainsi qu'un ensemble d'algorithmes permettant de transformer le code-octet pour améliorer les performances.

4.1. *La chaîne de production*

Le schéma de la figure 1 décrit la chaîne de production du code source Objective Caml jusqu'au fichier hexadécimal à transférer

sur le PIC. Le programme Objective Caml est d'abord compilé en code-octet grâce au compilateur standard *ocamlc*. Le code généré est ensuite nettoyé (par un algorithme d'élimination de code mort) et compressé. Le binaire obtenu est alors lié avec l'interprète et la bibliothèque d'exécution, tous deux écrits en assembleur. Le programme ainsi obtenu est assemblé et transféré sur le PIC.

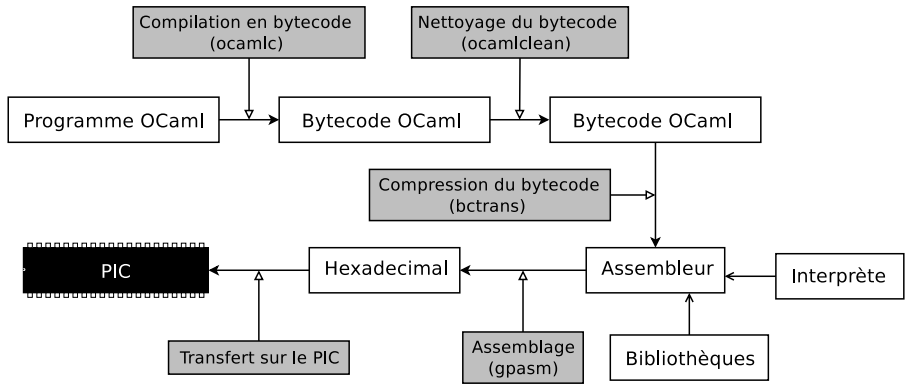


Figure 1 – Chaîne de production

4.2. *Représentation des données*

4.2.1. *Les valeurs Objective Caml*

Objective Caml utilise une représentation uniforme des données où chaque valeur est codée sur un nombre fixe d'octets. Une valeur peut être, soit une valeur immédiate (entier, booléen, etc.), soit l'adresse d'un bloc alloué dans le tas Objective Caml, soit une adresse en dehors du tas. Certaines primitives du langage (*e.g.*, `(Pervasives.compare: 'a -> 'a -> bool)` pour la comparaison en profondeur de valeurs) et le ramasse-miette (*garbage collector* ou GC par la suite) ont besoin de parcourir le graphe mémoire. Il est donc nécessaire de pouvoir distinguer, pour chaque valeur, s'il s'agit d'une adresse ou d'une valeur immédiate. Pour ce faire, nous utilisons le bit de poids faible des valeurs comme cela est fait dans `ocamlrun`. Les adresses étant paires, leurs bits de poids faible est

0. Il suffit donc de coder toutes les valeurs immédiates avec un bit de poids faible de 1. Par exemple, le booléen `false` est codé 0b1, le booléen `true` 0b11, et les entiers sous la forme $2 \times n + 1$.

Le nombre d'octets utilisé pour coder une valeur correspond habituellement à l'architecture de la machine réelle sur laquelle la machine virtuelle s'exécute (4 octets sur une architecture 32 bits et 8 sur architecture 64 bits). Cependant, les PIC utilisés ici ont une architecture 8 bits et la taille de la mémoire dynamique est de plusieurs ko. Il est donc impossible de représenter une adresse dans la mémoire dynamique sur un seul registre. De plus, avec des valeurs codées sur un nombre impair d'octets, les adresses n'auraient pas toutes été paires. Le choix qui a été fait est donc de coder les valeurs Objective Caml dans un PIC sur deux registres, soit 16 bits.

Ceci implique différentes restrictions. Notamment les entiers ne sont représentés que sur 15 bits. Ils sont donc compris entre -16 384 et 16 383. De plus, il est nécessaire de pouvoir coder une adresse de code dans une valeur, par exemple dans une fermeture. La taille du code-octet est donc limitée à 64ko.

4.2.2. *Les blocs*

Dans la machine virtuelle Objective Caml, toutes les valeurs présentes dans le tas sont regroupées dans des blocs. Un bloc est un tableau de données auquel on a ajouté un en-tête. L'en-tête contient habituellement trois champs : le champ *size* contenant la taille du bloc, le champ *color* utilisé par le gestionnaire mémoire pour marquer les blocs, et le champ *tag* donnant des informations sur le contenu du bloc. Certains blocs, comme les fermetures, contiennent des valeurs et doivent être parcourus par le gestionnaire mémoire. D'autres, comme les chaînes de caractères, contiennent de simples données *brutes* et ne doivent pas être analysés par le ramasse-miette. Le *tag* permet de distinguer les blocs à analyser des autres.

Dans la représentation choisie ici, les en-têtes sont codées sur deux octets et ne contiennent pas de champ *color*. Le ramasse-miette utilise une autre technique pour marquer les blocs. Celle-ci

est expliquée dans la section 4.4. Le champ *tag* est codé sur un octet comme dans la représentation standard pour des raisons de compatibilité. Le champ *size* est donc lui aussi représenté sur un octet, ce qui implique que les blocs contiennent au maximum 255 éléments.

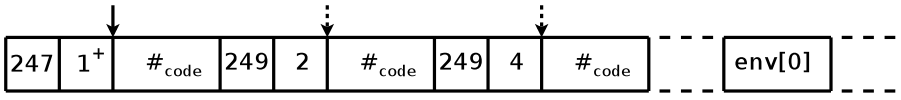
4.2.3. *Les fermetures*

Une fermeture est un bloc particulier servant à représenter une valeur fonctionnelle. Il y a deux types de fermetures : les fermetures simples et les fermetures récursives. Les fermetures récursives sont créées lorsqu'on déclare des fonctions récursives ou mutuellement récursives. Pour tout autre déclaration ou lors d'un appel partiel de fonction, une fermeture simple est créee.

Une fermeture simple est un bloc ayant pour *tag* 247, pour premier élément le pointeur de code, et pour autres éléments l'environnement. L'environnement correspond aux valeurs associées aux variables libres dans le corps de la fonction, et aux paramètres déjà passés à la fonction en cas d'appel partiel.

La structure des fermetures récursives est plus compliquée que celle des fermetures simples. Il s'agit de blocs contenant d'autres blocs. Les blocs internes sont des blocs ayant le *tag infixe* (249). Le bloc externe a le même *tag* qu'une fermeture simple (247). L'environnement est commun à toutes les fermetures (les internes + l'externe) et est stocké à la fin du bloc externe.

Voici la structure d'une fermeture récursive :



Le ramasse-miettes doit gérer les blocs internes d'une manière particulière. En effet, lorsqu'il arrive sur un bloc ayant le *tag* infixe (249), il ne doit pas simplement copier le bloc interne mais tout le bloc externe. Pour retrouver le bloc externe, il utilise le champ *size* du bloc infixe. Celui ci ne contient pas la taille du bloc infixe mais la distance entre le bloc interne et le début du bloc externe.

4.3. *Gestion des appels externes*

Les appels externes sont des appels, depuis le programme Objective Caml, à des fonctions écrites à l'origine en C. Ici, ces appels se font à des fonctions écrites en assembleur. Il reste toujours possible d'interfacer le programme Objective Caml avec n'importe quel langage. Pour ce faire, il est nécessaire de compiler les fonctions externes écrites dans ce langage en assembleur ainsi que d'avoir un contrôle sur le label généré par le compilateur au début du code assembleur de la fonction externe.

Comme cela a été expliqué dans la section 3, les fonctions externes sont associées à un identifiant entier défini dans la section PRIM de l'exécutable généré par `ocamlc`. L'appel à une fonction externe est compilé en une instruction particulière de la machine virtuelle nommée `ccall`. Cette instruction prend comme argument l'identifiant de la fonction externe et le nombre d'arguments qu'elle nécessite. L'instruction `ccall` consiste donc en un saut au code de la fonction externe, puis en le dépilement des arguments après le retour de la fonction.

Avant d'être transféré sur le PIC, le code-octet du programme subit des transformations décrites dans la section 5 pour réduire sa taille. En particulier, celles-ci convertissent la section PRIM en une table d'indirections. Ceci permet d'avoir un appel aux fonctions externes en temps constant : 9 cycles machine (7 instructions).

Depuis une fonction externe, il est possible de faire tout ce qui est possible depuis le monde Objective Caml, à savoir lancer une exception, appeler une fonction Objective Caml en utilisant sa fermeture, accéder aux variables globales, etc. Il est même possible d'accéder à la totalité de la mémoire en lecture et en écriture. Il faut donc prendre soin de ne pas rendre incohérent le graphe mémoire Objective Caml, sous peine de perturber le gestionnaire mémoire par la suite. Ceci est d'autant plus vrai sur un PIC car il n'y a pas de système d'exploitation et donc pas de vérification des accès mémoire. Dans ces conditions, un programme peut conti-

nuer à s'exécuter indéfiniement avec un comportement difficilement prévisible.

4.4. *Gestion automatique de la mémoire*

Le ramasse-miettes de la distribution standard d'Objective Caml combine différentes techniques [5]. C'est un GC incrémental à deux genérations [2]. Il utilise un `Stop & Copy` sur la génération jeune (GC mineur) et un `Mark & Sweep` incrémentiel sur l'ancienne (GC majeur). Enfin il peut être compactant si besoin est. Dans la réalisation pour PIC, nous avons choisi de coder un simple `Stop & Copy` qui ne cherche pas à optimiser la localité spatiale. En effet, un PIC n'ayant pas de cache, il n'y aucun intérêt à tenter d'améliorer cette dernière.

Il faut néanmoins faire attention à certains détails. Comme cela a été dit dans la section 4.2.3, les blocs infixes nécessitent un traitement particulier.

D'autre part, le ramasse-miette doit faire attention à ne pas confondre un pointeur de code et un pointeur dans le tas. Sur un ordinateur, ce problème ne se pose pas car la mémoire est en un seul morceau et les adresses dans le code ne peuvent pas correspondre à des adresses dans le tas. Sur un PIC, c'est différent car il y a plusieurs mémoires. Les pointeurs de codes sont des adresses dans la mémoire programme et les pointeurs dans le tas sont des adresses dans la mémoire vive. Les collisions sont donc *a priori* possibles. En fait, sur ces PIC, la mémoire vive est beaucoup plus petite que la mémoire programme. Il suffit donc de ne pas mettre de code-octet dans la mémoire programme à des adresses valides dans la mémoire vive. Cette mémoire n'est pas perdue car on peut mettre à la place le code de l'interprète et de la bibliothèque d'exécution.

De plus, lorsque le ramasse-miette copie un bloc, il stocke dans le premier champ du bloc original l'adresse de la copie. La question est : comment faire pour les blocs de taille 0 (habituellement appelés atomes) ? On remarque cependant que les atomes

ne contiennent pas de données. Et de plus, aucune valeur Objective Caml ne correspond à un atome de *tag* différent de 0. Il n'y a donc qu'un seul atome possible : celui ayant pour *tag* 0, pour *size* 0 et pas de données. Il suffit donc de l'allouer statiquement une fois pour toute en dehors du tas et de toujours utiliser le même.

Enfin, le ramasse-miette doit pouvoir marquer un bloc comme étant copié. Dans la représentation standard des blocs en Objective Caml, l'en-tête contient un champ *color* qui est là pour ça. Cependant, dans la représentation choisie ici (voir la section 4.2.2), l'en-tête des blocs ne contient pas de champ *color*. Or, comme il a été dit précédemment, aucun bloc présent dans le tas n'est de taille 0. Pour marquer un bloc dans le tas comme étant copié, il suffit donc de lui mettre le champ *size* de l'en-tête à 0.

5. Différentes transformations du code-octet

Pour installer un programme Objective Caml compilé en code-octet sur un PIC, il serait possible de l'écrire dans la mémoire programme à côté de l'interprète et de la bibliothèque d'exécution. L'interprète ZAM du PIC devrait alors effectuer exactement le même travail que l'interprète standard `ocamlrun`, à savoir désérialiser le tableau des variables globales, calculer les indices des fonctions externes et débuter l'exécution du code.

Il ne faut cependant pas oublier qu'un PIC a des ressources restreintes, tant au niveau vitesse de calcul qu'au niveau quantité de mémoire disponible. Une telle approche impliquerait inévitablement une initialisation atteignant facilement plusieurs dixièmes de secondes, et une forte contrainte sur la taille maximale des programmes. De plus, il est possible d'effectuer des analyses statiques sur le code-octet dans le but d'éliminer du code mort et des allocations inutiles.

Nous avons donc décidé de transformer l'exécutable généré par `ocamlc` avant de le transférer sur le PIC. Cette transformation consiste en une suite d'optimisations visant à améliorer le temps

d'initialisation, l'occupation de la mémoire programme, l'occupation de la mémoire vive et la vitesse d'exécution.

Cette passe supplémentaire sur le code-octet a de plus l'avantage de nous permettre de vérifier (partiellement) la compatibilité du programme avec la machine virtuelle. En particulier, des bornes des entiers utilisés doivent être compris entre -16384 et 16383 avec notre machine virtuelle 16 bits.

5.1. *Compression simple du code-octet*

Cette optimisation consiste en l'élimination, instruction par instruction, d'une partie importante des octets inutiles du code-octet. Les opcodes sont alors codés sur un octet au lieu de quatre, et chaque argument de chaque instruction est codé sur le nombre d'octets minimal (1 ou 2 selon les instructions).

Par exemple, l'instruction de la machine virtuelle `constint` permet de placer un entier litéral dans l'accumulateur. Son argument est un entier et est donc codé sur deux octets. De même, l'instruction `makeblock` permet d'allouer un bloc dans le tas. Elle prend un argument un tag et une taille qui sont tous les deux compris entre 0 et 255. Ils sont alors codés sur un octet chacun.

Cette compression simple permet de gagner un facteur 3,5 sur la taille occupée par le code-octet dans la mémoire programme du PIC, et par conséquent le même facteur 3,5 sur la vitesse de lecture du code par l'interprète[6]. La proportion moyenne entre le temps de lecture d'une instruction par rapport au temps total d'exécution de cette instruction étant de 6%, on en déduit qu'on gagne un facteur 1,15 sur la vitesse d'exécution totale. Le gain principal de cette optimisation concerne donc la taille maximale des programmes qu'il est possible de mettre sur un PIC, qui est alors multipliée par 3,5.

Par la même occasion, quelques optimisations ont été faites pour améliorer la vitesse d'exécution de l'interprète. Par exemple,

6. Rappelons que l'absence de cache (que ce soit pour la mémoire ou pour les instructions) dans les PIC permet de prédire aisément les temps d'exécutions.

les adresses de sauts ne sont plus codées par des adresses relatives (*offsets*), mais par des adresses absolues. De même, les entiers ne sont pas codés simplement en binaire mais plutôt dans leur représentation mémoire (sous la forme $2 \times n + 1$). Ceci permet d'éviter quelques calculs arithmétiques, et par conséquent quelques cycles machines à l'exécution.

5.2. *Optimisation du chargement des variables globales*

À l'initialisation du programme, le tableau des variables globales est habituellement désérialisé. Deux solutions ont été envisagées et codées pour tenter d'améliorer le temps d'initialisation. La deuxième s'est révélée être bien meilleure, tant sur le temps d'exécution que sur la mémoire programme utilisée.

La première solution est d'ajouter au début du programme du code-octet optimisé calculant le tableau des variables globales. Cette solution s'est révélée avoir un gain faible sur le temps d'initialisation par rapport à la solution naïve. De plus, elle apportait un grave problème : l'augmentation considérable de la taille du code-octet.

La deuxième solution est de précalculer, par ordinateur, avant le chargement du programme sur le PIC, l'état de la pile et du tas après la désérialisation du tableau des variables globales. Le contenu de la partie utilisée du tas et de la pile est alors stocké octet par octet dans la mémoire programme. À l'initialisation, l'interprète présent sur le PIC n'a qu'à copier un segment de mémoire programme dans la mémoire vive et démarrer l'exécution. Grâce à cette technique, le temps d'initialisation dans le pire cas (c'est-à-dire si le tas et la pile sont remplies à l'initialisation) sur un PIC ayant une horloge à 10MHz est au maximum de 2,5ms. Ce temps très faible est dû au fait que la mémoire dynamique des PIC de la série PIC18 est particulièrement petite : au maximum 4ko.

5.3. *Élimination de code mort (`ocamlclean`)*

Les optimisations décrites précédemment ne se sont pas révélées suffisantes pour permettre l'utilisation des objets Objective Caml sur PIC. En effet, la mémoire dynamique du PIC n'était pas suffisante pour supporter l'ensemble des fermetures créées à l'initialisation du module `Camlinternal00` et de ses dépendances. La mémoire était alors remplie de fermetures dont la majorité était inutilisée. Le codage d'un algorithme d'élimination de code mort a alors permis l'utilisation de la couche objet d'Objective Caml sur PIC.

Cette optimisation est basée, entre autres, sur une analyse statique du flot de données et du flot de contrôle. Elle consiste en l'élimination du code-octet de fonctions non utilisées ainsi que de la création des fermetures associées. Il s'en suit un gain, d'une part sur la taille du code-octet et d'autre part sur l'occupation de la mémoire dynamique. Grâce à cette optimisation, il est possible de mettre sur un PIC des programmes plus gros et allouant plus. De plus, comme le gestionnaire mémoire qui a été codé pour PIC n'est qu'un simple Stop & Copy, tous les blocs vivant sont copiés à chaque lancement du ramasse-miettes. Sa fréquence de lancement ainsi que son temps d'exécution sont eux aussi améliorés par l'élimination des fermetures non utilisées.

Le principal inconvénient de cette optimisation est qu'elle casse le chargement dynamique de modules. Ceci n'est pas une gêne pour le projet dans sa globalité car il est difficile de donner un sens à du chargement dynamique de code sur un PIC[7].

Le programme ainsi créé prend en entrée un exécutable standard pour la machine virtuelle Objective Caml, et génère en sortie un nouvel exécutable standard nettoyé. Il peut se décomposer en trois routines. Elles sont exécutées en boucle jusqu'à l'obtention d'un point fixe.

---

7. Les utilisateurs qui ne visent pas à embarquer leur code Objective Caml sur PIC peuvent s'en servir si le chargement dynamique ne leur est pas nécessaire. Cet outil (`ocamlclean`) est distribué dans une archive indépendante pour ne pas obliger à récupérer tout OCAPIC.

1) On analyse les variables globales à nettoyer : pour chaque bloc pointé par le tableau des variables globales, on teste les accès en lecture. Ceci est possible car ces accès sont explicites dans la machine virtuelle, ils ne peuvent se faire que par deux types d'instructions : `getglobal` (affectant l'accumulateur avec une variable globale) et `getglobalfield` (affectant l'accumulateur avec un champ d'une variable globale, cette instruction ne peut bien sûr être utilisée que pour les variables globales correspondant à des blocs). Les variables globales qui nous intéressent sont celles sur lesquelles aucun `getglobal` n'est effectué, seulement des `getglobalfield`. Pour ces variables, on détermine chaque champ non utilisé et on supprime son initialisation

2) On effectue ensuite une analyse du flot de données. Le but est de déterminer et d'éliminer du code-octet les instructions ayant créé les valeurs n'étant plus stockées dans les champ des variables globales nettoyées en 1. En particulier, on cherche à éliminer les instructions de création des fermetures (`closure` et `closurerec`) qui ne sont jamais stockées. Cet algorithme consiste en la création d'un graphe de dépendances entre les instructions, puis en un parcours de ce graphe pour déterminer les instructions dont le seul effet a été de calculer des valeurs qui ne sont jamais lues.

3) Enfin, on effectue une analyse standard du flot de contrôle pour déterminer les blocs de code non-atteignables lors de l'exécution. De tels blocs n'existaient pas dans le code original, mais sont apparus lorsque l'on a supprimé des instructions de création de fermetures à l'étape 2.

Cet algorithme a été entièrement codé en 1800 lignes de code Objective Caml. Il a été aussi testé avec `ocamlrun` sur les compilateurs `ocamlc` et `ocamlopt` et semble fonctionner correctement.

6. Environnement de développement

L'ensemble des algorithmes décrits dans les sections précédentes ont été implantés et forment une application complète permettant de programmer les PIC de la série PIC18 en Ob-

jective Caml. Pour utiliser cette application, il faut prendre en compte certains points. En particulier, la bibliothèque standard a été en partie modifiée : certaines fonctionnalités ont été retirées et d'autres ajoutées pour correspondre aux besoins spécifiques de la programmation sur PIC. De plus, pour permettre le debogage des programmes Objective Caml sur PIC, il a été codé une sorte de simulateur. Enfin, les ressources d'un PIC étant particulièrement restreintes, il faut maîtriser les performances de la machine virtuelle.

6.1. *Une nouvelle bibliothèque standard Objective Caml pour PIC*

Une partie de la bibliothèque standard n'a pas été portée sur PIC. Parmi les différents modules, certains ont été copiés tels quels et d'autres ont été restreints, recodés, ou supprimés. En particulier, toutes les fonctions concernant les entrées/sorties standards ont été suprimées des modules `Pervasives` et `Buffer` car elles n'ont pas vraiment de sens sur un PIC. Le tableau 3 indique ce qui a été fait de chaque module de la bibliothèque standard.

Modules copiés	Modules modifiés	Modules non-portés
`Array, ArrayLabels, CamlinternalLazy, CamlinternalMod, Char, Hashtbl, Int32, Int64, Lazy, List, ListLabels, MoreLabels, OO, Queue, Set, Sort, Stack, StdLabels, String, StringLabels`	`Buffer, CamlinternalOO, Gc, Map, Obj, Pervasives, Random, Std_exit, Sys`	`Arg, Complex, Callback, Digest, Filename, Format, Genlex, Marshal, Nativeint, Parsing, Printexc, Printf, Scanf, Stream, Weak`

Tableau 3 – Portage des modules de la bibliothèque standard

De plus, comme cela avait été dit dans la section 2.2, l'un des problèmes lors du portage d'un langage de haut niveau sur PIC est de permettre au programmeur d'accéder aux ressources bas niveau, sans casser la sécurité ni les principes de programmation associés au langage haut niveau. Une bonne illustration de ce pro-

blème est la gestion des registres spéciaux du PIC depuis le monde Objective Caml. Les registres spéciaux d'un PIC désignent les ports et les registres de configuration des interfaces internes du PIC. Depuis l'assembleur, on lit et on écrit librement dans ces registres. La question est de donner au programmeur Objective Caml le moyen d'accéder à ces registres aussi facilement qu'en assembleur. La solution qui a été adoptée est d'utiliser les types sommes d'Objective Caml dont on maîtrise la représentation mémoire. Un type somme représente l'ensemble des registres spéciaux, et un autre l'ensemble des bits des registres spéciaux. La représentation mémoire des valeurs de ces types correspond dans un cas à l'adresse des registres, et dans l'autre à l'adresse des registres concaténée avec les masques d'accès aux bits. La manipulation des registres spéciaux et de leurs bits est alors possible depuis le monde Objective Caml en passant par des fonctions externes.

La programmation sur PIC n'a pas les mêmes domaines d'application que la programmation sur ordinateur. Il est donc intéressant de fournir au programmeur Objective Caml un ensemble de bibliothèques pour lui permettre de se rapprocher des domaines d'applications standards des PIC. Par exemple, il est habituel d'utiliser un PIC pour contrôler un petit afficheur LCD. Le PIC doit alors appliquer des protocoles assez bas niveau pour communiquer avec l'afficheur. Il est intéressant de fournir au programmeur Objective Caml une interface haut niveau lui permettant d'utiliser facilement un afficheur LCD. De plus, on aimerait avoir une certaine généricité vis-à-vis de l'afficheur, ou plus simplement vis-à-vis de là où est branché l'afficheur. Pour résoudre ce problème, les foncteurs sont une solution assez agréable. Nous avons donc codé un module supplémentaire dans la bibliothèque standard nommé `Lcd`. Voici un exemple de programme utilisant ce module :

```
module Aff = Lcd.Connect (struct
  module Pic = Pic18f4620
  let is_8_bits = true (* mode de comm. (bus de données) *)
  let e  = Pic.RD0     (* déf. des pattes de commandes *)
  let rw = Pic.RD1
  let rs = Pic.RD2
  let port = Pic.PORTC (* port pour le bus de données *)
end)
open Aff
```

```
10 let () = init ()             (* initialisation de l'afficheur *)
11 let () = config             (* configuration de l'afficheur *)
12           ~mode:Cursor_right(* sens de déplacemt du curseur *)
13           ~disp:On          (* affichage visible *)
14           ~cursor:Off       (* curseur invisible *)
15           ~blink:Off        (* cuseur non clignotant *)
16           ~lmode:Two        (* affichage sur 2 lignes *)
17           ~font:F5x8        (* police de caractères 5x8 px *)
18 let () = write_string "Hello world" (* ^^ *)
```

6.2. *Un simulateur de PIC pour les programmes Objective Caml*

Le débogage d'applications pour PIC est un problème assez délicat. En effet, lorsqu'un programme s'exécute sur un PIC, à moins d'utiliser des débogueurs in-situ[8], il est impossible de suspendre l'exécution pour observer le contenu de la mémoire. De plus, un PIC n'est pas forcément connecté à une interface permettant de communiquer facilement avec un humain (comme par exemple un afficheur LCD). Il est alors impossible de tracer l'exécution du code en insérant des « printf ».

Il existe des émulateurs de PIC comme *gpsim*, et la plupart des environnements de développement intégrés comme *MPASM* ou *PIC simulator IDE* en contiennent. Cependant, il n'est pas très pratique d'utiliser de tels simultateurs pour déboguer les programmes Objective Caml sur PIC. En effet, ces simulateurs parcourent le code de la machine virtuelle, et il est difficile de faire le lien entre les instructions simulées par le déboguer et le code source Objective Caml.

Nous avons donc codé un outil premettant au programmeur Objective Caml de simuler l'exécution de ses programmes sur un PIC. Comme on peut le voir sur la chaîne de production décrite dans la figure 1, le programme Objective Caml est d'abord compilé en un exécutable standard pour ordinateur. Cet exécutable est en

8. Technique de mise au point pour des applications embarquées sur microcontrôleur consistant à le connecter à un ordinateur et exécuter le code dans le microcontrôleur. L'ordinateur peut alors contrôler partiellement le flux d'exécution du programme dans le microcontrôleur et accéder à sa mémoire.

fait lié à une bibliothèque d'exécution particulière qui intercepte les appels aux fonctions externes de lecture et d'écriture dans les registres spéciaux du PIC. Lors de l'exécution sur un ordinateur de cet exécutable, le processus lancé ouvre différentes fenêtres permettant de visualiser les actions du PIC. Il est alors possible de simuler les interactions du PIC avec différents composants électroniques. Par exemple, lorsque l'on exécute le programme correspondant au code Objective Caml de la section 6.1 grâce à la commande suivante, on peut visualiser une fenêtre représentant la réaction d'un afficheur LCD aux actions du PIC comme sur la figure 2. Comme le programme s'exécute sur un ordinateur, on peut utiliser les outils standard de débogage (par exemple `ocamldebug`) pour tracer le code Objective Caml.

```
./hw 'ocapic_lcd_simulator 16x2 e=RD0 rs=RD2 rw=RD1 bus=PORTC'
```

Figure 2 – Capture d'écran de la simulation d'un PIC connecté à un afficheur LCD

7. Étude de performances

7.1. *Performances générales du système* OCAPIC *sur le PIC18F4620*

La vitesse d'exécution moyenne des programmes sur un PIC exécutant 10 000 000 instructions machine par seconde est de 400 000 instructions de code-octet par seconde. Autrement dit, il faut en moyenne 25 cycles machine pour exécuter une instruction de code-octet. De plus, le PIC perd en moyenne 7,5 cycles machine dans la gestion de chaque opcode. Ceci implique que la proportion du temps d'exécution des instructions utiles par rapport au temps d'exécution de la gestion du code-octet est de 70%.

La taille de la mémoire programme du PIC18F4620 est de 64ko. La taille du binaire associé à l'interprète est de 4 954o. La taille du binaire associé à la bibliothèque d'exécution est de 4 254o. Par conséquent, la taille maximale du code-octet que l'on peut mettre dans ce PIC est de 55ko, ce qui représente 86% de la mémoire totale.

La mémoire vive du PIC18F4620 est de 3 968o. Elle est partagée entre la pile d'exécution, le tas, des registres réservés à la machine virtuelle (39o), et les registres spéciaux (128o). La proportion entre la taille de la pile et la taille du tas est définie à la compilation. Par défaut, la taille de la pile a été arbitrairement fixée à 174 niveaux. Comme l'algorithme de gestion mémoire est un Stop & Copy, seule la moitié du tas est utilisable. La taille du tas est donc ici de 1792o.

7.2. *Calcul de nombres premiers avec allocations de listes chaînées*

Nous proposons d'observer le temps d'exécution d'un programme calculant la liste chaînée des n premiers nombres premiers. Le principal site d'allocation mémoire est situé dans la définition de la seconde fonction (`npremiers_aux`), il s'agit du constructeur de listes (`::`). Nous pouvons ainsi prédire le nombre d'éléments alloués dans le tas. Ce nombre correspond en fait à l'allocation du résultat attendu : les n premiers nombres premiers dans une liste chaînée. La liste allouée occupe alors $n \times (2+2+2) = 6 \times n$ octets, ce qui fait 1500o pour une liste à 250 éléments. Étant donné que le tas du PIC18F4620 est limité à 1792o, n=250 est arbitrairement une bonne valeur.

```
1 let rec npremiers n =
2  if n = 0
3  then []
4  else npremiers_aux (n-1) [2]
5 and npremiers_aux n r =
6  if n = 0
7  then r
8  else npremiers_aux(n-1) (premier_suivant(List.hd r+1) r :: r)
9 and premier_suivant n l =
```

```
10  if est_premier n l
11  then n
12  else premier_suivant (n+1) l
13 and est_premier n = function
14  | [] ->
15    true
16  | e::tl ->
17    if n mod e = 0
18    then false
19    else est_premier n tl
20
21 let () =
22   for i = 1 to 80 do
23     ignore(npremiers 250)
24   done
```

Nous comparons à titre indicatif les temps d'exécutions obtenus sur un ordinateur moderne (Linux 2.6.32 32 bits, processeur *Intel Atom Z520 1,33GHz* et mémoire vive *1Go DDR2 667MHz*) et un PIC18F4620 cadencé à 10MHz, en moyenne sur 5 exécutions. Dans les deux cas, le même compilateur `ocamlc` est utilisé (distribution Objective Caml 3.12.0).

Nous obtenons alors 3,74 secondes pour l'ordinateur à 1,33GHz et 472 secondes sur le PIC18F4620 à 10MHz. On observe un facteur 126 entre les deux temps d'exécution.

7.3. *Jeu du solitaire*

Le test classique de performances du jeu du solitaire a été effectué dans les mêmes conditions que celles présentées dans la section précédente. Nous avons mesuré les temps d'exécutions sur ordinateur moderne grand public et sur le PIC18F4620.

Nous obtenons cette fois-ci 7,2 secondes pour l'ordinateur à 1,33GHz et 812 secondes sur le PIC18F4620 à 10MHz. On observe un facteur 113 entre les deux temps d'exécution.

7.4. *Implantation d'un jeu de stratégie, algorithme minimax*

Pour valider notre approche, nous avons implanté un jeu de plateau à deux joueurs. Le premier joueur est un humain utilisant une interface composé de boutons poussoir. Le second joueur est artificiel (« incarné » par un programme exécuté sur le PIC). Le programme indique au joueur les actions et l'avancement du jeu par l'intermédiaire d'un afficheur LCD.

La réactivité du PIC est satisfaisante malgré les multiples déclenchements du ramasse-miette lors de l'exécution de l'algorithme de réflexion. En effet, le temps de calcul est en moyenne d'une demie seconde et toujours inférieur à deux secondes. Un humain expérimenté a une certaine difficulté à gagner, ce qui laisse penser que les algorithmes implantés sont satisfaisants.

Le jeu a été implanté en 667 lignes de code Objective Caml. Une implantation de ce jeu en assembleur PIC ou dans un langage de bas niveau obligeant à gérer manuellement le peu de mémoire disponible sur un PIC aurait été très ardue voire quasi impossible.

8. Travaux connexes

L'environnement de développement MPLAB fourni avec les microcontrôleurs PIC propose un assembleur (MPASM), un éditeur de liens (MPLINK) et un simulateur pour tester et mettre au point un exécutable. Il est possible de programmer dans des langages de plus haut niveau. On trouve principalement des sous-ensembles de C et de Basic interfacés avec MPLAB ou autonomes.

Il y a trois démarches dans les autres langages proposés : compilateur natif (comme pour C), interprète (comme pour Basic), compilation vers une machine abstraite puis interprétation du code-octet comme présenté dans cet article.

Pour les compilateurs natifs, on trouve des variations autour du langage Pascal comme Pic micro Pascal [11] qui s'intègre aux outils de MPASM/MPLINK de MPLAB tout en proposant un IDE propre. Un autre langage plus éloigné de Pascal, Jal [14] a pro-

posé dès 2004 un compilateur libre. Une nouvelle version, Jal2, a été dévelopée par une communauté toujours active.

Pour les interprètes, on rencontre principalement des implantations du langage Forth qui ont l'avantage d'être petites et donc de pouvoir tenir sur un microcontrôleur. FlashForth [9] est un système Forth autonome implanté sur PIC18F qui apporte un interprète et un compilateur/chargeur, les deux embarqués sur le microcontrôleur. Il existe par ailleurs des compilateurs Forth qui permettent de charger le code comme PicForth [13]. La gestion des ports et des interruptions est en règle générale assez facile à manipuler en Forth comme le montre l'exemple de compilation sur le site de PicForth.

Nous ne sommes pas les premiers à utiliser l'approche machine abstraite pour cibler l'architecture PIC. En effet, la conception d'une machine virtuelle Scheme pour PIC a déjà été entreprise au sein des projets PICBIT [4] et PICOBIT [12] pour un sous-ensemble du R4RS. PICBIT adapte aux contrôleurs PIC un environnement Scheme très compact. La machine abstraite écrite en C est traduite par un compilateur C pour PIC. PICOBIT étend cette démarche en y adjoignant un compilateur C, appelé SIXPIC, spécifique aux PIC et permettant d'obtenir un meilleur code en particulier pour la machine abstraite. Comme indiqué lors de l'analyse des performances, l'écriture directe de l'interprète ZAM et de la bibliothèque d'exécution en assembleur PIC, combinée à la compaction du code, donne de très bonnes performances. En cela les démarches PICOBIT et OCAPIC divergent, la première privilégiant la portabilité sur d'autres architectures et la deuxième l'efficacité sur PIC.

On retrouve cette approche machine abstraite pour le langage Java avec la Plateforme Java Card [10]. Néanmoins même en limitant fortement le langage Java (en particulier la gestion automatique de la mémoire, l'absence de certains types (entiers 64 bits) et de contrôle comme les threads), et en écrivant dans un style impératif, on obtient toujours un code important qui ne peut tenir sur des microcontrôleurs actuels.

9. Conclusion

Cette réalisation permet de programmer des microcontrôleurs PIC en Objective Caml. Elle est prête à l'utilisation et distribuée[9] sous la licence libre CeCILL-B. L'implantation dans un langage de très bas niveau de la machine virtuelle Objective Caml sur PIC permet d'obtenir de très bonnes performances. Le côté paradoxal de notre démarche est qu'en simplifiant la programmation pour PIC en utilisant Objective Caml, on arrive à faire exécuter des programmes plus importants, qui peuvent donc contenir une part algorithmique non négligeable, de manière plus sûre grâce au typage statique et à la gestion automatique de la mémoire tout en restant très efficace. Cette démarche ouvre des perspectives intéressantes pour le portage d'un langage de haut niveau comme Objective Caml sur des appareillages ayant de faible ressources, tant en mémoire qu'en capacité de calcul.

Il reste alors à enrichir la bibliothèque pour PIC, afin de pouvoir tirer davantage profit de ces microcontrôleurs ainsi que des périphériques usuellement associés à ces derniers. Ces enrichissements pourront intégrer aussi des extensions sur les modèles de concurrence liés au développement d'applications interactives et réactives pour PIC, basées sur l'environnement Objective Caml.

Un autre avantage de cette démarche « machine abstraite » est de pouvoir raisonner sur du code-octet plutôt que sur l'assembleur PIC, soit pour prouver des propriétés par analyse statique, soit pour vérifier à base de tests d'autres propriétés du programme comme l'expérience du projet Couverture[10] l'a montré dans le cadre du développement certifié.

La machine virtuelle Objective Caml qui a été codée dans ce projet fonctionne pour les microcontrôleurs de la série PIC18. Le portage de cette machine virtuelle sur les séries antérieurs des PIC (PIC10 à PIC16) semble difficile voire impossible. En effet, la quantité de registre et de mémoire programme risquerait d'être insuffisante. Sur les séries de PIC plus avancées (PIC24 à PIC32),

9. http://www.algo-prog.info/ocaml_for_pic/
10. http://www.projet-couverture.com/

le portage de cette machine virtuelle est très rapide car les assembleurs des séries récentes sont presque compatibles avec celui de la série PIC18. Il faut néanmoins remarquer que le code de la machine virtuelle pourrait alors être optimisé pour tirer parti des fonctionnalités apportées par les nouvelles séries de PIC. Enfin, le portage sur d'autres types de microcontrôleurs (*e.g.*, AVR, Atmel, Philips) demanderait un travail conséquent car les architectures internes et les assembleurs sont différents.

Enfin, cette expérience de portage d'Objective Caml sur PIC prouve qu'il est possible d'utiliser des langages de haut niveau pour programmer des architectures exotiques ayant de très faibles ressources.

Références

[1] Bigonoff. La programmation des PIC par Bigonoff, cinquième partie. Migration vers 18F : mode d'emploi, 2010. `http://www.abcelectronique.com/bigonoff/`.

[2] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'Applications avec Objective Caml*. O'Reilly, 2000. `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML`.

[3] Xavier Clerc. *Cadmium*, February 2010. `http://cadmium.x9c.fr/distrib/cadmium.pdf`.

[4] Marc Feeley and Danny Dubé. Picbit : a Scheme system for the PIC microcontroller. In *Scheme Workshop*, pages 7–15, November 2003.

[5] Richard E. Jones. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[6] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, February 1990.

[7] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system (release 3.11) : Documentation and user's manual*. INRIA, November 2008.

[8] Microchip. *PIC18F2525/2620/4525/4620 Data Sheet*, 2008. http://ww1.microchip.com/downloads/en/devicedoc/39626b.pdf.

[9] Mikael Nordman. Flashforth. http://flashforth.sourceforge.net/.

[10] Oracle. *Java Card 3.0.1 Platform Specification*, May 2009. http://www.oracle.com/technetwork/java/javacard/specs-jsp-136430.html.

[11] Philippe Paternotte. *Pic Micro Pascal V1.4 : User Manual*, July 2010. http://www.pmpcomp.fr.

[12] Vincent St-Amour and Marc Feeley. Picobit : A Compact Scheme System for Microcontrollers. In *International Symposium on Implementation and Application of Functional Languages (IFL'09)*, pages 1–11, September 2009.

[13] Samuel Tardieu. Picforth programmer manual. http://www.rfc1149.net/devel/picforth.

[14] Wouter van Ooijen et al. *Jal (not ?) Just Another Language*, May 2004. http://jal.sourceforge.net/manual.